A USER-FRIENDLY PROGRAMMING FRAMEWORK FOR WIRELESS SENSOR NETWORKS

A Thesis
Presented to
The Academic Faculty

by

Shruthi Ravichandran

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
August 2014

A USER-FRIENDLY PROGRAMMING FRAMEWORK FOR WIRELESS SENSOR NETWORKS

Approved by:

Raheem A. Beyah,
Advisor and Committee Chair
School of Electrical and Computer Engineering
Georgia Institute of Technology

Yang Wang
School of Civil and Environmental Engineering
Georgia Institute of Technology

Yusun Chang
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: 15 May 2014

This thesis is dedicated to my mother and late father. I never imagined I would be saying late, Appa. You've dedicated your entire life to my well-being, happiness and education. Nothing will ever be enough to thank you for all that you've given me. Thank you Amma for being so very strong and supportive, even during the hardest times of our lives. You are simply amazing.

ACKNOWLEDGEMENTS

I would first, like to thank my advisor, Dr. Raheem Beyah for giving me this opportunity to grow as a researcher and for his unwavering confidence in me; Dr. Selcuk Uluagac, for his constant support and encouragement; and to both of them, for their great ideas and for giving me creative freedom in the development of this work.

I would like to thank Dr. Yusun Chang and Dr. Yang Wang for serving on my thesis reading committee and for their helpful suggestions towards the improvement of my work; particularly, Dr. Yang Wang's group for helping make this work user-friendly.

Finally, I would like to show appreciation to everyone in the Communications Assurance and Performance (CAP) group for making this past year a great working experience. A special mention to Ram, for being a great mentor and for always taking time off to help me with issues in this research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Wireless sensor networks (WSNs) are increasingly being used for monitoring physical environments in lieu of tethered monitoring systems. Being power efficient and wirelessly accessible, WSNs find applications in a variety of domains like health, structural and climate monitoring systems. Despite such diverse use cases, more than often, WSNs are used by researchers with basic or no prior programming experience. Consequently, more time is spent learning to program the sensors than collecting and analysing domain-specific data. To cater to this generic user base, in this thesis, a multi-platform user-friendly programming framework for WSNs has been developed. This framework provides: 1) a visual network comparison tool that analyzes packet traces of two networks to generate a juxtaposed visual comparison of contrasting network characteristics, 2) a scripting language based on the TinyOS sensor network platform that aims at reducing code size and improving programming efficacy, and 3) a visual programming tool with basic sensor drag-and-drop modules for generating simple WSN programs. These tools were also developed to serve as a gentle introduction to the WSN programming environment for middle and high school students. In the absence of resources (sensors), the framework also allows programmers to verify program functionality by remotely simulating and verifying program behaviour in the OMNeT simulation environment. In this thesis, the network comparison, scripting and visualization tools are discussed in detail along with sample test scenarios created with real sensors.

CHAPTER I

INTRODUCTION

Recent advances in sensors and wireless technology have led to the proliferation of wireless sensor networks in heath, structural and environmental control and monitoring systems. TinyOS [1] is one of the most widely used sensor network platforms to develop and program a myriad of applications in these sectors [2, 3]. Despite its large user base, novice application developers face a very steep learning curve in writing TinyOS applications in nesC [4] to perform basic data collection and distribution. A lot of systems that provide application-specific modules as add-ons to the main WSN program have been developed [5, 6]. These modules implement the most commonly used sensor network applications like data collection, link estimation, etc., that can be added as single line function calls to the main program. However, they lack the flexibility of a simple scripting language that allows users (even novices) to experiment with their code and develop something beyond these add-on modules. The scripting tool developed in this work concentrates on this feature while still being relatively easy to use even for novices.

There are many tools that help the user to debug their programs by monitoring and visualizing packets in the WSN. These tools either capture live network packets [7] or visualize packets passively from a packet trace [8]. While these tools are very useful for visual debugging, they run short in delivering useful network statistics to the user such as packet load in the network, average delay between nodes, etc. The network comparison tool aims to provide these statistics and at the same time allows users to compare two WSNs and visually verify differences in the performance of the two networks in terms of link characteristics.

Efforts to introduce programming to middle or high schoolers are very common [9, 10]. These tools are extremely user friendly in that they teach children programming concepts

such as loops and if-else statements in a very simple graphical user environment. Nevertheless, there are not many tools that attempt to do the same with sensors. Sensors are very attractive with their multi-color LEDs, light and temperature sensors and radio interface. With these interfaces, we can teach the students similar programming concepts, but with readily observable outputs in the LEDs or through RF packets that can be visualized from packet traces. The visual programming tool for WSNs aims to close this gap and offer a simple graphical user interface (GUI) for K-12 students to interact easily with sensors and progress rapidly into programming sensors with the scripting language.

This thesis aims at developing and integrating all these tools into a single framework [8] that will allow users (experts, novices, middle schoolers) to develop code with the scripting language or with the graphical programming tool, test code via visualization and verify performance by observing network statistics.

1.1 Research Objective

The objective of this thesis is to develop and integrate the following tools on top of the PROVIZ framework [8]: 1) a network comparison tool that compares two WSNs from packet traces based on their respective link characteristics, 2) a scripting language based on TinyOS programming concepts that allows for easy programming of sensor network concepts and, 3) a graphical programming tool with a drag and drop interface for developing simple sensor network programs.

Framework Highlights

The framework consists of the three aforementioned tools: network comparison, scripting language, and graphical programming.

The network comparison tool parses the packet traces provided by user and computes the differences between two networks based on weights given by the user for pre-determined link characteristics. On comparing the two traces, the tool informs the user if the networks match based on a threshold matching percentage. The tool also shows the two network topologies next to each other with missing and extra links (with respect to the first network) indicated and the link properties of each link are displayed when the link is right-clicked.

This tool is discussed in more detail in chapter 3 along with screenshots.

The scripting language inherits the best of TinyOS concepts such as modularity and split-phase operations and at the same time is simple enough for a novice programmer to develop programs for data collection, monitoring and distribution. The scripting language uses simple C/C++ abstractions of TinyOS concepts as described in chapter 4. This chapter also describes how WSNs can be simulated in situations when procuring resources (sensors) for testing the WSN programs is difficult. For example, many students do not have access to these sensors and would be capable of developing programs, but are unable to test them on real sensors. In those cases, the framework interfaces with the OMNeT simulator [11] to simulate the WSN and users can verify program functionality from the output generated by OMNet.

The graphical programming tool has rows of actions or functions for each sensor. Basic input (light, temperature sensor), output (LEDS, Radio) and operation (On, Off) modules can be dragged and dropped onto these rows to create actions. Each sensor is given a node ID and once all actions are created, the user can drag and drop the sensor onto a canvas and connect to the other sensors via radio as required. This tool is described in more detail in chapter 5.

These tools are built on the PROVIZ framework such that all the tools in this framework can be used in sync with each other to visualize, program and verify WSNs as shown in chapter 6.

CHAPTER II

LITERATURE REVIEW

The literature in WSNs covers many diverse topics, the relevant ones which can be subdivided into works on WSN analysis, scripting languages for sensor network programming and graphical programming of WSNs.

2.1 WSN Analysis

When deploying multiple sensor networks, it might be imperative to debug the deployments, obtain network characteristics and compare them for differences and irregularities. Our approach to the comparison of two WSN deployments based on link characteristics obtained from packet traces is novel. At the same time, there are a lot of tools that monitor the WSN and obtain network properties such as topology, packet loss, etc. The following are some of the relevant tools developed in this area.

MOTE-VIEW [12] is a user interface software developed by MEMSIC [13] for monitoring and visualizing a WSN deployment. It acts as a health monitoring tool that queries a database server (populated by packets from nodes) to retrieve node status and configuration of all nodes in the network. It is also used for visualizing node qualities such as throughput, bandwidth, link quality, and congestion (obtained from server) in color codes in a topology chart.

Livenet [14] is a tool developed to analyze a sensor network reconstructed from packet traces. The tool uses multiple packet sniffers and merges all the packet traces to get a more detailed account of the WSN behavior. From this merged packet trace, network characteristics such as topology, bandwidth usage and packet loss are analyzed and plotted.

SNIF [15] and Sympathy [16] are tools developed for network debugging and fault finding within the WSN. While Sympathy debugs control information sent by sensors along with regular traffic to a sink node, SNIF uses a deployment support network [17], which is a network of secondary sensors, used specifically for debugging to find faults in the network.

NetViewer [7] is a real-time visualization and monitoring tool. It collects data from a sink node and based on packet format specified by user, parses the packets, visualizes the packets, displays payload information and demonstrates topology of the network. Other than the topology, this work does not provide any other important network detail to the user.

Octopus [18] is also a WSN monitoring, visualization and additionally, a control tool. Besides visualizing the WSN in real-time, Octopus can also reconfigure the sensor network by sending out short control messages. Code for this tool has to be installed in the sensor along with the sensor network program to avail its functions or, the program and packet structure have to be developed according to Octopus specifications. Despite being an efficient monitoring and control tool, it is limited by its requirement for code provided by Octopus to run on sensors and its inability to extract network characteristics from packet traces.

All the tools discussed above are efficient monitoring and/or fault detection tools for WSNs, but they are different in that they do not provide specific network characteristics to the user such as average size of packet, average packet delay, etc. or have the capacity to compare two networks .

2.2 WSN Scripting Languages

TinyOS [1] is a widely used operating system for sensors. It uses nesC [4], a language based on C for developing WSN applications. Although nesC is very well established and can be used to write many complex applications, the process is still very tedious involving writing multiple files (configuration and implementation files) and many lines of code. The following papers have developed scripting languages that reduce the complexity of developing applications in nesC.

SNACK [6] is a kit consisting of a configuration language, a service library and compiler for programming WSNs. With SNACK, WSN users spend minimal time to put together an application by using its in-built service library that consists of commonly used functions such as routing, sensing, etc. The compiler will compile this application written in the

configuration language to nesC. The syntax of the configuration language is a little hard for novices and SNACK does not give users the flexibility to add user-defined functions to its library or reuse program blocks like the scripting tool.

Ceu [19] is a higher level programming language for C and nesC that reduces program size and the complexity of developing applications in TinyOS. But, despite reducing code size and programming time considerably, the scripting language developed is still not very user-friendly and requires quite a bit of familiarizing. The language is not part of an integrated framework like PROVIZ where code can be developed and simulated. Additionally, it does not implement TinyOS concepts such as modularity thereby, relinquishing the code and reuse property of TinyOS modules and interfaces.

DSN [5] is a declarative language and compiler for fast and efficient sensor network programming. Similar to SNACK, it also offers a library of add-on modules to be added to programs written in Snlog, a dialect of Datalog [20]. The DSN compiler compiles this program to configuration and module files in nesC. Since the DSN is based on a data querying language, novices again, face a steep learning curve.

The scripting languages discussed above, despite having very short code size, are a little difficult to comprehend. They also don't support the code use and reuse property (modularity) of TinyOS. The scripting tool tries to keep the code size to a minimum, and at the same time ensures flexibility for users to develop their own code and reuse them across files.

2.3 WSN Visual Development

The relevant visual development tools for WSNs can be broadly classified into application development environments and visual programming tools. Application development environments are GUIs for creating TinyOS configuration files by visually wiring together component and module files from the TinyOS library; while visual programming tools use visual blocks for creating sensor network programs.

Viptos [21] is a graphical application development tool for TinyOS-based WSNs. It allows users to develop TinyOS programs by constructing block and arrow diagrams of

TinyOS components. This is to remove the programming complexity of locating and wiring various components together from the TinyOS library. TOSDev [22] is also a similar application development tool for TinyOS. Like Viptos, this tool also offers graphical editing of wiring diagrams along with a source code editor similar to Eclipse [23].

RaPTEX [24] is an integrated code development, simulation and emulation environment that offers TinyOS protocol stack development with wiring and component connection diagrams similar to Viptos; creation of topology and simulation of network level performance in OMNeT; and, emulation of node-level behavior using Avrora [25] as the underlying-emulator. With this integrated environment, the tool allows easy customization of existing TinyOS protocol suites. It is also, a platform for testing programs before deploying a WSN. The current work is different in that it offers tools for post-deployment analysis of a WSN like the network comparison tool and a simpler scripting-based programming environment for novice users, instead of nesC-based code development.

WISDOM [26] is a platform-independent visual programming tool that uses a modular approach to programming. Users can create new modules written in C and connect them graphically with connectors and the tool will generate the platform dependent code for the program (in this case, nesC). This work is not very well documented. It does not reveal how sensor-relevant code (timer, radio and led function calls) are declared in C. Also, this tool does not seem to allow creation of independent modules in an application as they are limited by connectors that chain modules together.

SEAL [27] offers a visual programming and standard programming language (as an alternative to nesC) for WSNs. The visual programming language abstracts the SEAL programming language using Google Blocky [28]. The programming language is, as evaluated by them, easy for novices. However, by steering away from TinyOS, it loses some advantages offered by TinyOS such as split-phase operations and code reuse.

TinyInventor [29] is a visual programming tool that adopts the OpenBlocks [30] visual programming language to create a TinyOS relevant programming environment for WSNs. The programs are a collection of functional blocks created by users with drag and drop components. The blocks are compiled to generate nesC code for sensors. The tool is

simple and easy to use and is similar to Scratch that also uses OpenBlocks. However, the main disadvantage is that the tool is not part of an integrated system like PROVIZ where visualization, programming and simulation can be done from the same system.

The visual programming tools discussed above are simple to use. But, they lack the pictorially descriptive drag-and-drop modules that the visualization tool developed in this thesis offers and are not part of an integrated framework like PROVIZ where programs can be built, simulated or programmed and visualized, all from a single platform.

CHAPTER III

NETWORK COMPARISON TOOL

The network comparison tool is a tool designed to compare two networks based on their respective link characteristics. The tool can be used by researchers who would like to compare two current deployments or two past deployments. The tool can also be employed in the education of programming concepts in the middle and high school levels. The educators can assign homework to students to program a WSN according to specifications. These programming assignments can be created by expert programmers along with solution packet traces. This eliminates the involvement of educators who might not be well versed with programming and WSN concepts. The tool, can then be used to compare the solution packet trace of the prescribed WSN and the trace from the WSN simulated or programmed by students. Students can be graded by percentage similarity to the solution network and can themselves, visually identify and learn the differences in the two WSNs.

3.1 Design

The network comparison tool extracts packet information (arrival time, the IEEE 802.15.4 header [31] and the packet payload) from .psd trace files generated by the TI SmartRF Packer Sniffer [32]. The tool writes this packet information to XML first for easier access and manipulation of packet characteristics. The parsed XML file is read and a graph structure with nodes and links is created. The tool uses a weighted-average approach to match the packet traces based on weights given by a user. We have identified five typical link characteristics that a user tries to identify in a sensor network: number of packets sent, average packet delay, number of bytes sent, average packet size, and periodicity of packets sent. The weights (in percentages of the total score) for each of these link properties are obtained from the user before comparing the traces. The weights are taken into account for calculating individual scores for each packet trace. The trace file to be compared is referred to as the problem and the trace to be compared against is the solution.

Table 1: Link score calculation for a graph.

PROPERTIES	WEIGHTS in (in "%")	PROPERTY SCORE
Number of Packets	30	80
Average Packet Delay	0	0
Number of Bytes	0	0
Average Packet Size	0	0
Periodicity	70	100
WEIGHTED AVERAGE LINK SCORE		94

A score for each of the properties is calculated based on the percentage accuracy of the value of the property with respect to the value in the corresponding link of the solution trace. Once this one-to-one mapping for all the properties is done, a weighted average of the link score is calculated. If the score passes a threshold 'N', the link gets a score of 1, else, it gets a 0. If N percent of the total number of links have a score of 1, trace matching/similarity is reported. Once this process of mapping the two traces is done, the two graphs are displayed along with the link properties to help visually identify the difference in properties, links and nodes. Additionally, the link quality indicator (LQI) values of each of the links are displayed with the properties. Table 1 shows the calculation of a link score.

After initiating this comparison and calculating network similarity, the two networks are plotted using the Graphviz graph library [33] in Qt [34] and the percent network similarity is reported. The two networks are shown next to each other along with missing nodes/links color coded in red and extra nodes/links in blue.

3.2 Sample Scenario

The above mentioned packet trace comparison tool was tested on real sensors (MicaZ) and WSNs to ensure robustness and efficiency. We set up five MicaZ sensors in two slightly different network configurations. Packet trace (.psd) files of the two networks were obtained from the TI SmartRF Packet Sniffer. These traces were compared by the packet comparison tool and the graph comparison UI was generated using Graphviz.

Figure 1 shows the user interface (UI) for obtaining the weights for each of the five link characteristics.

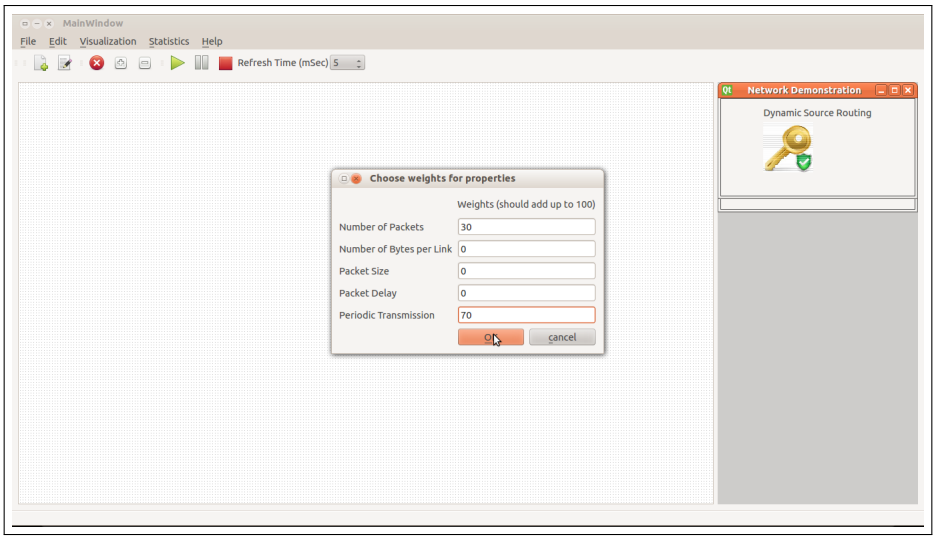


Figure 1: UI illustrating the weights obtained from the user.

Figure 2 shows the two traces' topology in comparison to each other. The extra nodes/edges are drawn in blue and the missing nodes/edges in red. On right-clicking each link, the properties of that link and the link in the other graph are displayed respectively. When a missing or extra edge is right-clicked, only link properties of the link in that graph are reported.

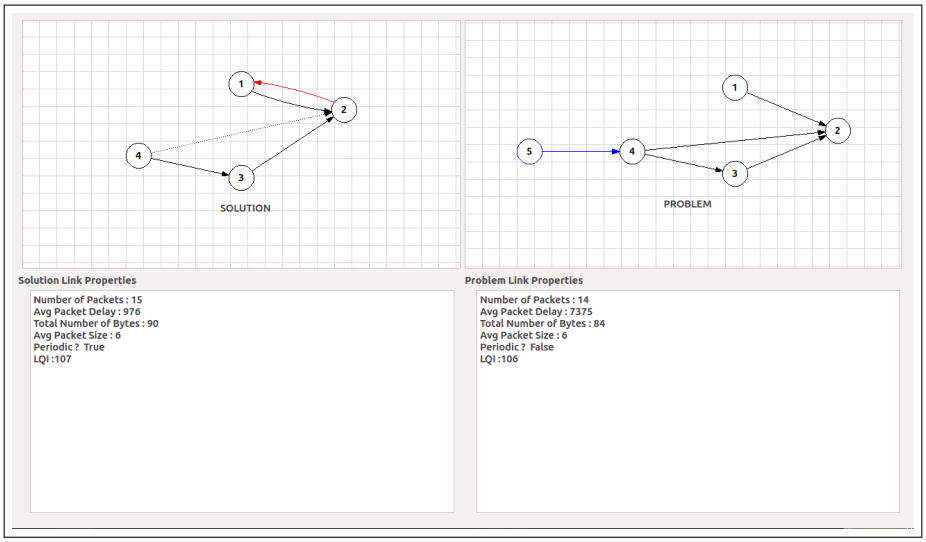


Figure 2: Comparison of graphs when right-clicking a common link.

Figure 3 shows the comparison of graphs when a missing edge is right-clicked on the solution graph.

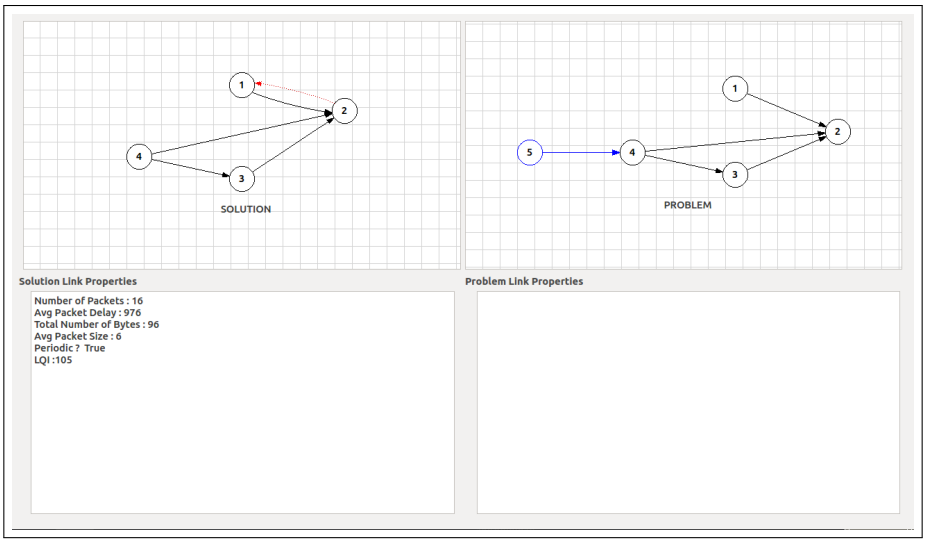


Figure 3: Comparison of graphs when a missing edge is right-clicked.

Figure 4 shows the dialog box pop-up after the comparison is done. It reports if the traces match and the percent matching.

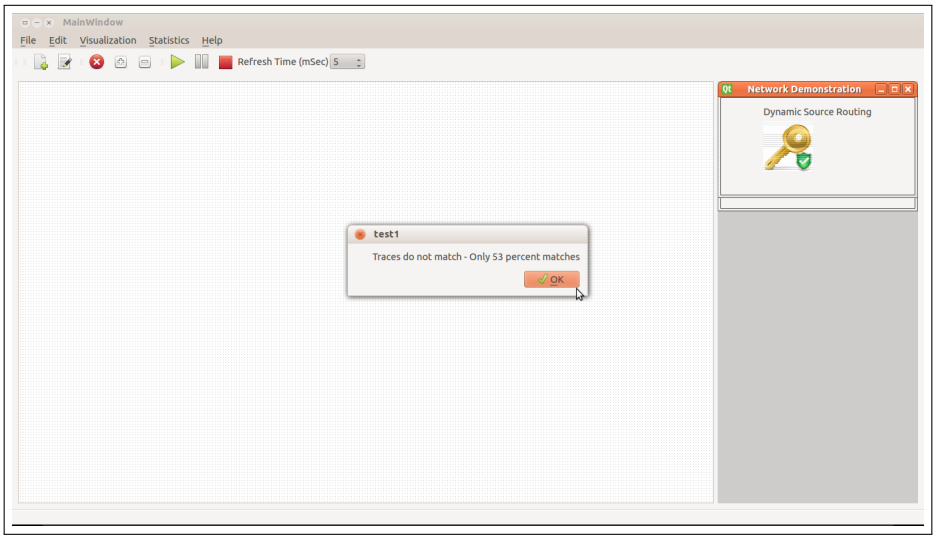


Figure 4: Dialog box showing the result of the packet trace comparison and the percentage matching.

CHAPTER IV

TINYOS SCRIPTING LANGUAGE

TinyOS [1] is an operating system for WSNs that uses nesC [4], a C-based language for developing sensor network applications. The system is widely used, but its main shortcomings for programmers are the complicated nesC language itself and the multiple files that have to be written for a single application. The scripting language was developed to overcome these difficulties by having a simpler, user-friendly language based on C/C++ concepts with single-line commands for commonly used functions, and the ease of writing the entire application in a single file.

4.1 Application Development with nesC

TinyOS requires the creation of two files for building an application - a configuration file and an implementation file. The configuration file contains wiring of modules (i.e., functions) and interfaces that are used and provided by this application. The implementation file is where the actual sensor behavior is written. TinyOS operations are split-phase operations, meaning, function calls for operations like send packet, start timer, etc. that take too long to execute are not blocking calls. Instead, the calls are returned and when the operation finishes execution, callbacks (events) are generated. For a novice developer, these concepts are hard to grasp.

Figures 5 and 6 are snapshots of the configuration and implementation files of a sample *RadioCountToLeds* application in TinyOS. The application starts a counter that updates itself every n milliseconds and sends out a packet containing the counter value. When a packet is received, the value of the counter sent is read from the payload of the packet and the least significant bits (LSBs) are displayed on the LEDs. Every node running this application sends out counter values and receives counter values of other nodes in the network and flashes them to the LEDs. The snapshot of the implementation file is only about half the entire program. As is apparent, this is information overload for a novice

user, even if the user has basic programming knowledge.

```
configuration RadioCountToLedsAppC {}
implementation {
  components MainC, RadioCountToLedsC as App, LedsC;
  components new AMSenderC(AM_RADIO_COUNT_MSG);
  components new AMReceiverC(AM_RADIO_COUNT_MSG);
  components new TimerMilliC();
  components ActiveMessageC;

  App.Boot -> MainC.Boot;

  App.Receive -> AMReceiverC;
  App.AMSend -> AMSenderC;
  App.AMControl -> ActiveMessageC;
  App.Leds -> LedsC;
  App.MilliTimer -> TimerMilliC;
  App.Packet -> AMSenderC;
}
```

Figure 5: Configuration file for the application.

```
module RadioCountToLedsC @safe() {

  uses interface Leds;
  uses interface Boot;
  uses {
    interface Receive;
    interface AMSend;
    interface Timer<TMilli> as MilliTimer;
    interface SplitControl as AMControl;
    interface Packet;
  }
}
implementation {

  message_t packet;

  bool locked;
  uint16_t counter = 0;

  event void Boot.booted() {
    call AMControl.start();
  }

  event void AMControl.startDone(error_t err) {
    if (err == SUCCESS) {
      call MilliTimer.startPeriodic(250);
    }
    else {
      call AMControl.start();
    }
  }

  event void AMControl.stopDone(error_t err) {
    // do nothing
  }

  event void MilliTimer.fired() {
```

Figure 6: Part of the implementation file for the application.

4.2 Scripting Language

In the previous section, a sample TinyOS application was discussed and the difficulty in programming with nesC was elaborated. Taking this into account, the design of the scripting language was influenced by the following factors: 1) eliminating the writing of a configuration file and the wiring involved, 2) abstracting interfaces and making it simpler to create and reuse code and, 3) developing a language that can be scaled to include more abstracted functions and complex applications.

With these factors in mind, a C-based scripting language was chosen as C's basic principles are simple and easy to learn and because many WSN users, even novices, may have some basic embedded programming knowledge. While there are some scripting language specific function calls and include headers, the rest of the program written with this language follows only C. Therefore, users can port any C program to this script by simply modifying only those functions that are sensor-specific. Figure 7 shows the *RadioCountToLeds* application written in C language.

```
struct test_counter_msg {
    int count;
};
 int counter = 0;

void INITIAL() {
    startPeriodicTimer_1(1000);
}
event timerTimeout_1() {
    counter = counter + 1;
    createPacket(packet1, test_counter_msg,
counter);
    sendPacket(packet1, 20);
}
event receivePacket(test_counter_msg* msg) {
    if(msg->count & 0x1)
        led0On();
    else
        led0Off();
    if(msg->count & 0x2)
        led1On();
    else
        led1Off();
    if(msg->count & 0x4)
        led2On();
    else
        led2Off();
}
//main implementation cannot be changed
void main() {
    INITIAL();
}
```

Figure 7: *RadioCountToLeds* in the scripting language.

In the TinyOS application, a boot-up sequence of actions specified by the user is the starting point of the program. This boot-up sequence is called when the sensor boots, but this function call is not visible to users. To avoid this confusion, a main() function is added prior to the script that has a single function call to INITIAL(). The users will, then, have to define actions in INITIAL() as the boot-up sequence. The scripting language tries to expose the split-phase operation of function calls in TinyOS as much as possible while still trying to remain tractable. Apart from calling the timer and radio interfaces to start the timer or send packets, the user will have to define what has to be done in the callbacks (events) that are generated. This TinyOS concept was retained in the scripting language to offer more flexibility to the users in developing their application. Packet payloads are

defined in structs for easy access of payload information and callback functions are defined with event return type as a keyword.

Figures 8 and 9 describes the TinyOS property of providing and using interfaces as a C++ abstraction of static classes. Creating an interface (a sequence of TinyOS commands and events) involves an interface name and the name of the module that implements the interface. This is the only way TinyOS allows the user to develop code in one file and use it across many files. This can be abstracted to the concept of a static C++ class (interface) and the include file in which the static functions are defined (module). Functions to be exported are defined as static class functions and the program that uses these functions, calls them as regular static functions (without a class object) and includes the name of the program that implemented them.

```
void INITIAL() {
      //implementation
}
void Routing::getStateTables()
{
      //implementation
}
void Routing::getHopCount()
{
      //implementation
}
//main implementation cannot be changed
void main() {
      INITIAL();
}
```

Figure 8: Sample script that implements an interface in the form of static class functions.

```
#include "Route"

void INITIAL() {
      //implementation
      Routing::getStateTable();
}
//main implementation cannot be changed
void main() {
      INITIAL();
}
```

Figure 9: Script that uses Routing functions implemented in the previous script.

4.3 OMNeT simulation

The script generates nesC code that can be either programmed onto a WSN or simulated in the OMNeT simulator [11]. OMNeT is a C++ based, modular network simulator. The advantages in using OMNeT is that it allows heterogeneous network simulation where each node in the network can run different applications, unlike TOSSIM [35] that can simulate

only homogeneous networks. TinyOS applications can be simulated in OMNeT using NesCT [36]. NesCT is a language translator that takes nesC programs as input and generates OMNeT code that can be simulated in the OMNeT environment. But NesCT only takes version 1 of TinyOS as input, while the latest release of TinyOS is version 2. The scripting language generates TinyOS-2.x code. To convert TinyOS-2.x code to TinyOS-1.x code, a parser was developed in Python. There are a few differences in TinyOS-2.x and TinyOS-1.x code and the parser simply parses these variations in code and generates TinyOS-1.x code that can be fed into the NesCT language translator. On simulating in OMNeT, the packets from the simulator output are sent to PROVIZ for visualization.

CHAPTER V

VISUAL PROGRAMMING TOOL

The visual programming tool can be used to introduce youngsters to programming concepts with the help of sensors. The tool aids in creating a heterogeneous WSN with multiple sensors running custom programs. It has simple, visually descriptive drag-and-drop icons to create a sensor network program and connect sensors via radio. With this tool, reading temperature or light sensor, playing with the LEDs, sending messages over the air and setting up timed actions in sensors become much simpler. This tool generates code in the scripting language described in the previous chapter. The tool will be a stepping stone for learning a new language such as C, and sensor network programming. They can visually see how their drag-and-drop icons translate to the scripting language and learn there onwards.

5.1 Design

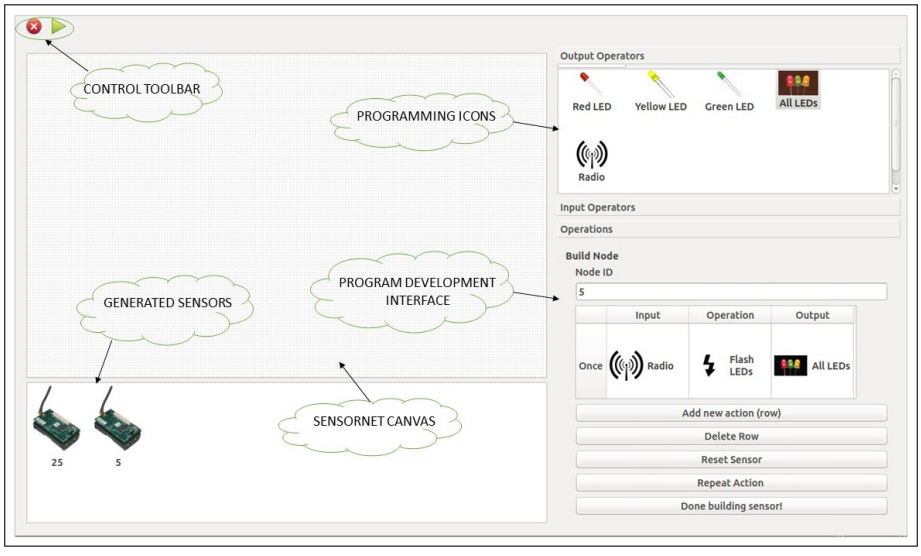


Figure 10: User interface of the visual programming tool.

The visual programming tool was developed in Qt [37] on top of the PROVIZ framework. Qt is an open source, multi-platform, C++ based framework for developing graphical user interfaces (GUIs). The user interface (UI) of the tool has five sections: 1) *Control Toolbar* - for generating code and resetting the UI, 2) *Programming Icons* - drag-and-drop icons to create sensor programs, 3) *Program Development Interface* - for creating action sequence, setting timer values, node ID, etc., 4) *Generated Sensors* - list of sensors developed in the program development interface and, 5) *Sensor Network Canvas* - canvas for creating a sensor network environment with sensors generated by the program development interface. The different sections are discussed below.

5.1.1 Control Toolbar

The control toolbar has options to reset the whole interface and start fresh and to generate code once the WSN is created in the *Sensor Network Canvas*.

5.1.2 Programming Icons

This interface has three sets of draggable icons for programming - input operators, output operators and operations. These icons have to be dragged and dropped onto the table to create an action sequence. The following three tables describe the function of each icon.

Table 2: Input operators.

Input Operator	Name	Function
	Light Sensor	Reads the light sensor value
	Temperature Sensor	Reads the temperature sensor value
	Radio	Reads payload value sent via radio

Table 3: Operations.

Operation	Name	Function
	Turn Off	Turns off the LED
	Turn On	Turns on the LED
	Send To	Sends input value over to radio
	Flash LEDs	Flashes LSB to output LEDs

Table 4: Output operators.

Output Operator	Name	Function
	All LEDs	Operation routed to all LEDs
	Green LED	Operation routed to Green LED
	Red LED	Operation routed to Red LED
	Yellow LED	Operation routed to yellow LED
	Radio	Input value sent over radio

5.1.3 Program Development Interface

This interface has a dynamic table where each row represents an action sequence in a sensor. For example, a sequence can be reading a sensor value and flashing the value using LEDs or another action sequence such as turning on the red LED. The three columns represent

input operators, operations and output operators, respectively. An action sequence can be repeated with a timer by selecting the row to be repeated and specifying the timer value in milliseconds. Action rows can be deleted or added dynamically. Once the sensor is built, a node ID is given to the sensor and the sensor is generated. That is, the program for this sensor is generated in the scripting language in the background and visually, the sensor is added to the *Generated Sensors* list.

5.1.4 Generated Sensors

Once a sensor is built in the *Program Development Interface*, the sensor is added as an icon to this list along with node ID. As the user continues building programs for other sensors, sensors are accumulated in this list. These sensors can be dragged and dropped onto the *Sensor Network Canvas*.

5.1.5 Sensor Network Canvas

Sensors can be dragged and dropped onto this canvas to create a heterogeneous WSN. Nodes can be removed from the canvas by right-clicking a node and selecting the delete option. Nodes can be connected to each other via radio by right-clicking and connecting to a list of nodes in the canvas. By selecting this option, if the source node has an output *Radio* operator, packets are sent to the destination node ID specified.

5.2 Test Scenario

The following figures are screenshots of the tool in action for developing a two-node heterogeneous WSN.

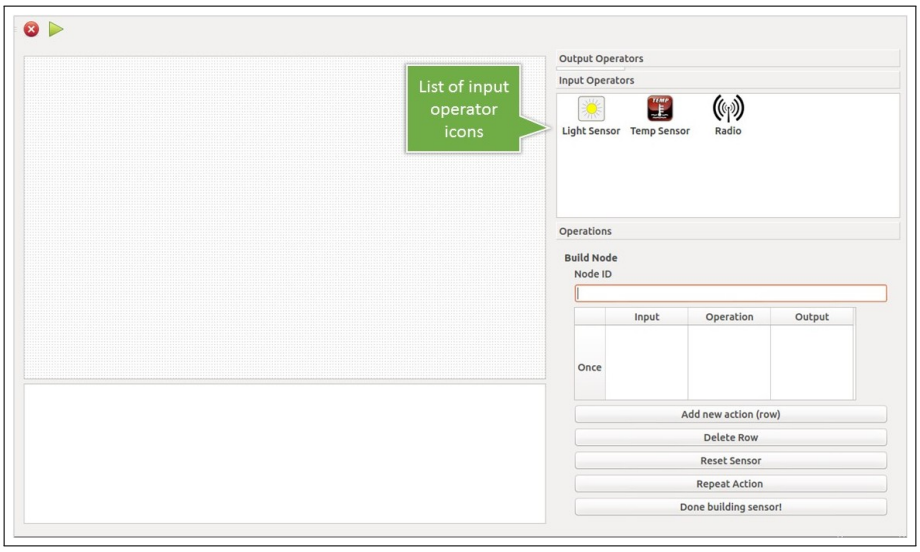


Figure 11: List of input operator icons in the *Programming Icons*.

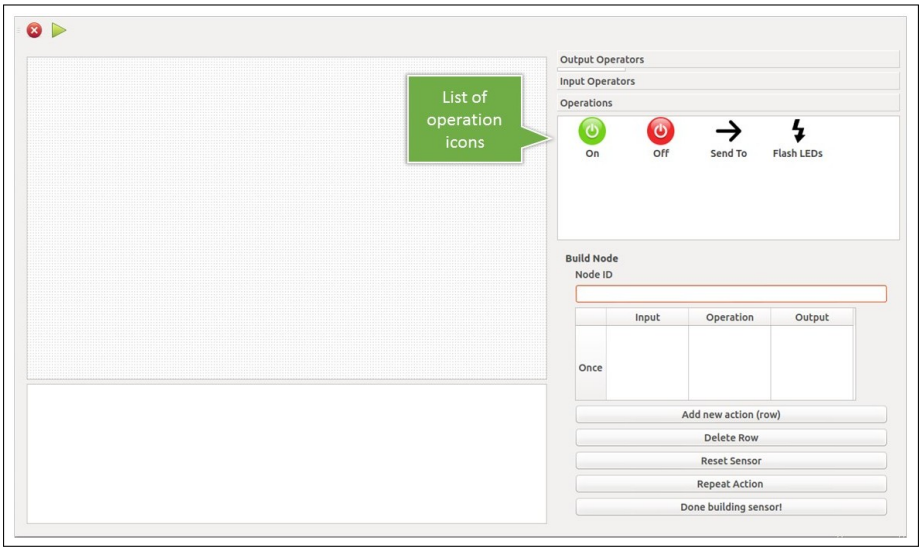


Figure 12: List of operation icons in the *Programming Icons*.

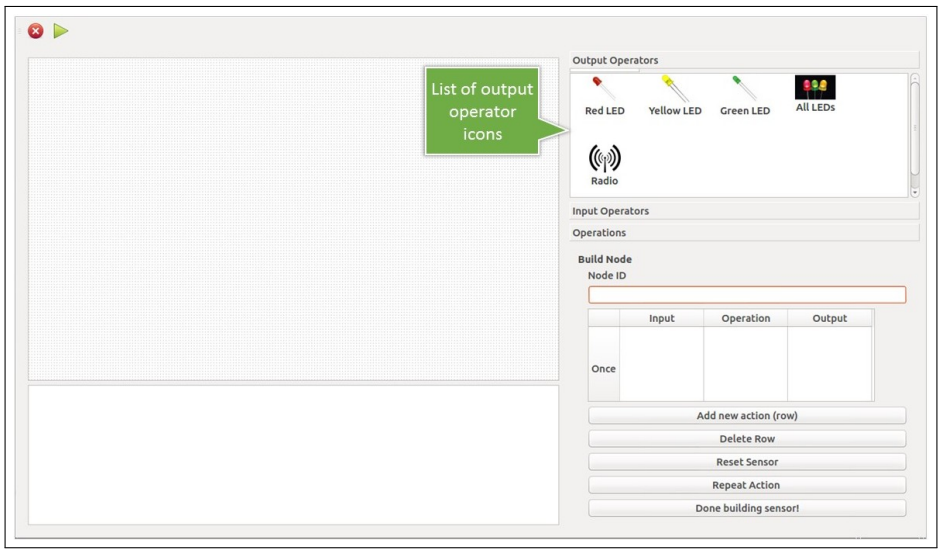


Figure 13: List of output operator icons in the *Programming Icons*.

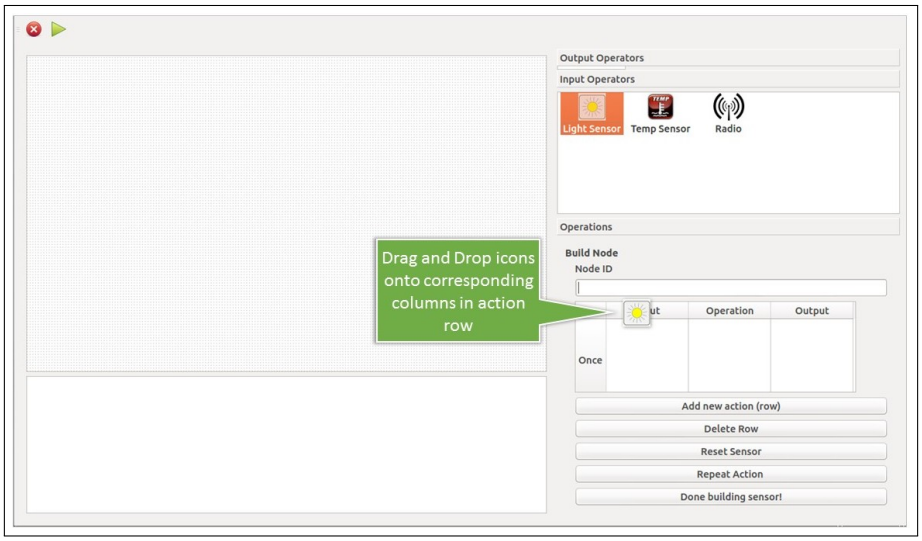


Figure 14: Dragging of an icon to the *Program Development Interface*.

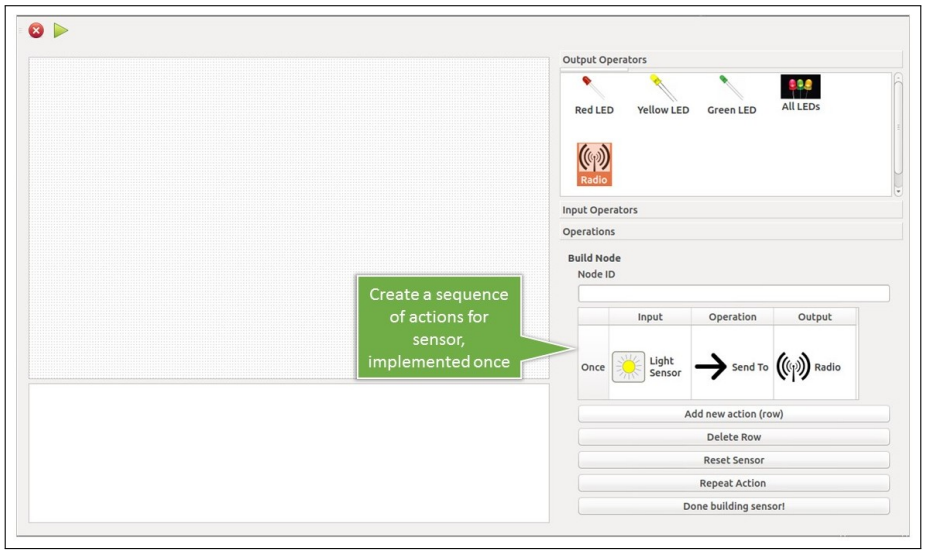


Figure 15: Creation of an action - reading light sensor value and sending over radio.

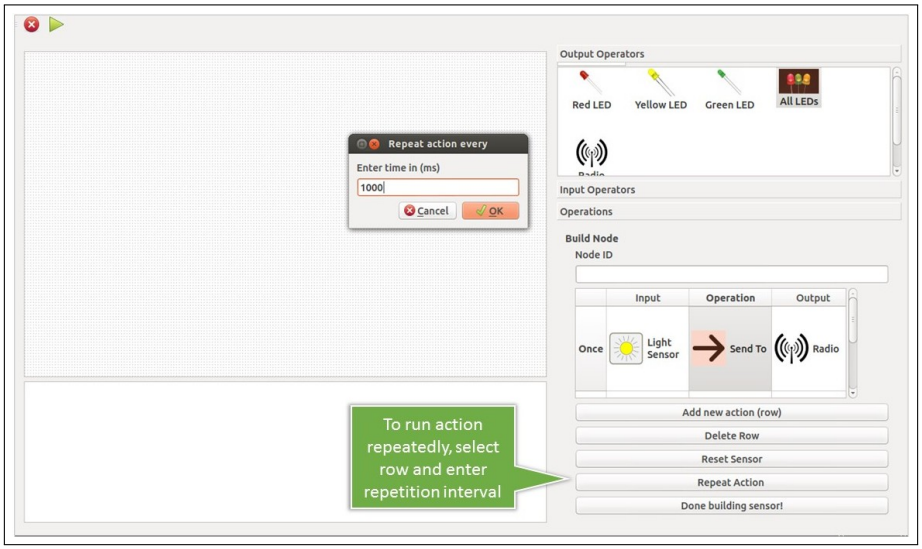


Figure 16: Repetition of an action sequence.

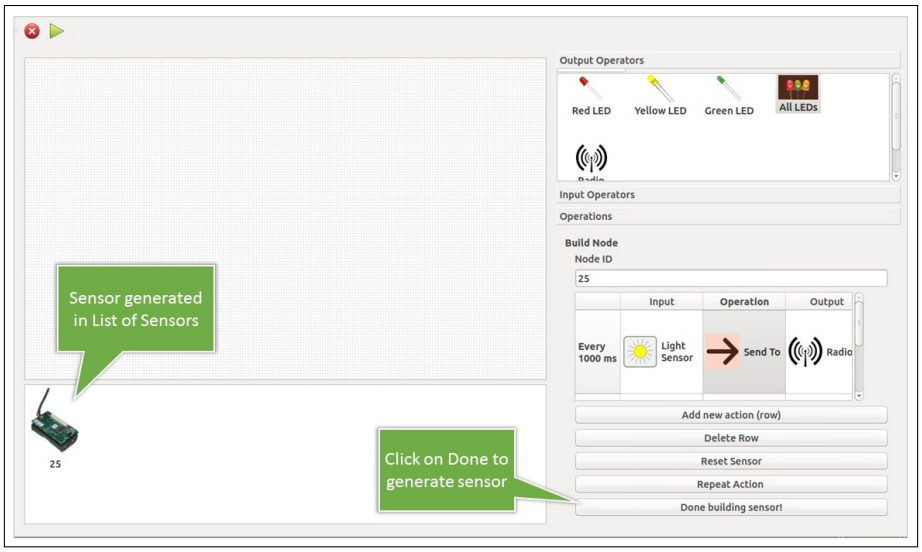


Figure 17: Building and generation of a sensor in *Generated Sensors*.

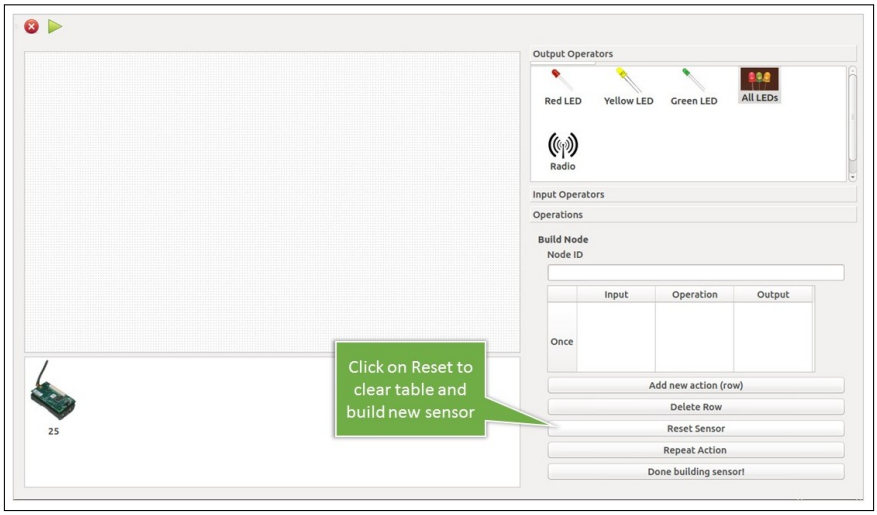


Figure 18: Reseting of the *Program Development Interface* to build new sensor.

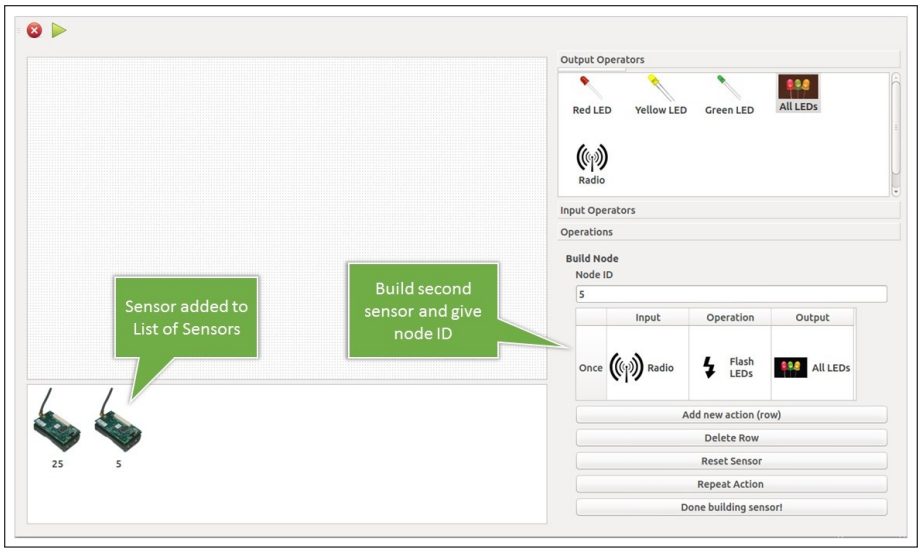


Figure 19: Addition of a second sensor to the framework.

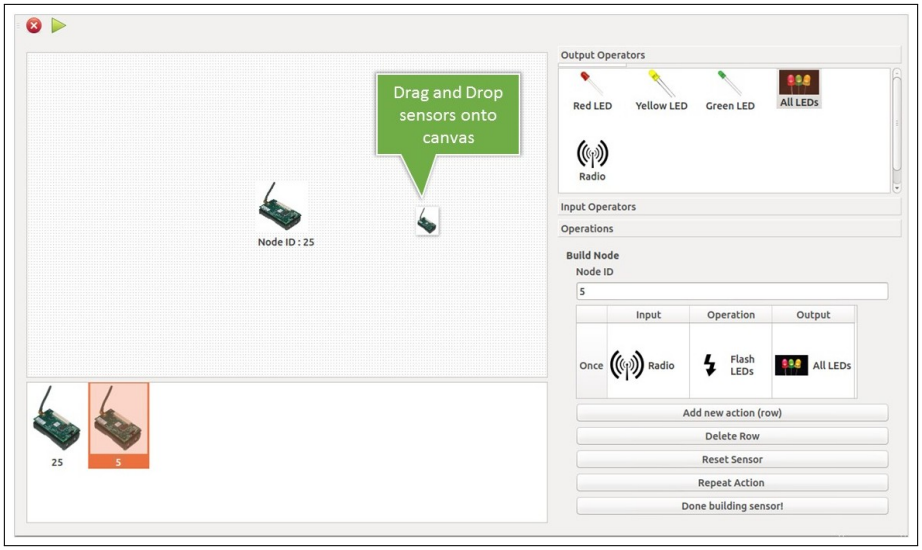


Figure 20: Dragging and dropping of sensors onto the *Sensor Network Canvas*.

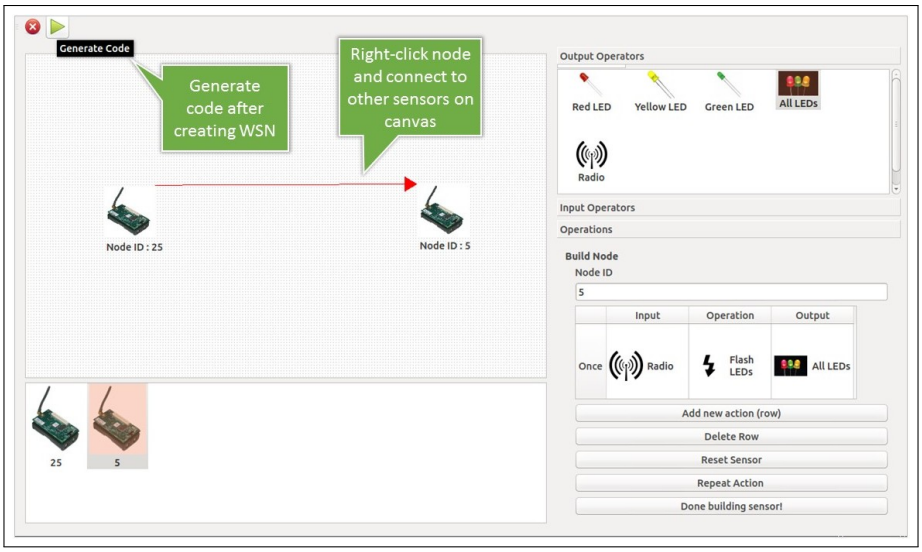
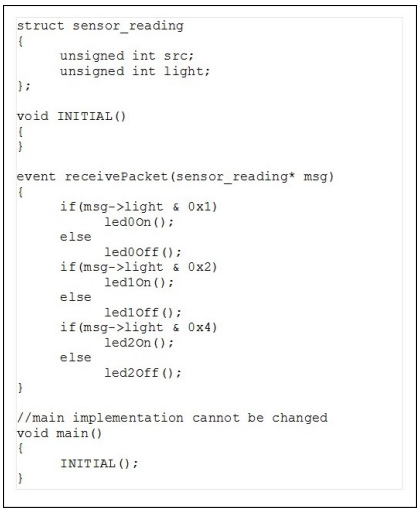


Figure 21: Creation of a WSN and generating code for the same.

The code generated in the scripting language for the two sensors are as shown in figures 22 and 23.

```
struct sensor_reading
{
      unsigned int src;
      unsigned int light;
};
unsigned int lightInfo;

void INITIAL()
{
      startPeriodicTimer_1(1000);
}

event timerTimeout_1()
{
      startSenseLight();
}

void startSenseLight()
{
      lightInfo = senseLight();
      createPacket(packet1, sensor_reading, nodeID, lightInfo);
      sendPacket(packet1, 5);
}

//main implementation cannot be changed
void main()
{
      INITIAL();
}
```

Figure 22: Program for node 25 in the scripting language.



```
struct sensor_reading
{
      unsigned int src;
      unsigned int light;
};

void INITIAL()
{
}

event receivePacket(sensor_reading* msg)
{
      if(msg->light & 0x1)
            led0On();
      else
            led0Off();
      if(msg->light & 0x2)
            led1On();
      else
            led1Off();
      if(msg->light & 0x4)
            led2On();
      else
            led2Off();
}

//main implementation cannot be changed
void main()
{
      INITIAL();
}
```

Figure 23: Program for node 5 in the scripting language.

CHAPTER VI

PROVIZ OVERVIEW

The three tools discussed in the previous sections are built on the PROVIZ framework [8] and thereby, reap the benefits of having a real-time and passive visualization tool working in sync with them.

The PROVIZ framework originally used MCL [38] as a scripting language for code generation. It supports live visualization from sniffers located across the network that are synchronized to a Network Time Protocol (NTP) server; and passive visualization from .psd packet trace files generated from the TI SmartRF Packet Sniffer [32]. Programs generated with PROVIZ are simulated in OMNeT and the packet traces generated from OMNeT are sent to the visualization tool. Figure 24 depicts the old framework in detail, with the addition of three new modules - network comparison tool, scripting language tool, visual programming tool and an updated OMNeT interface.

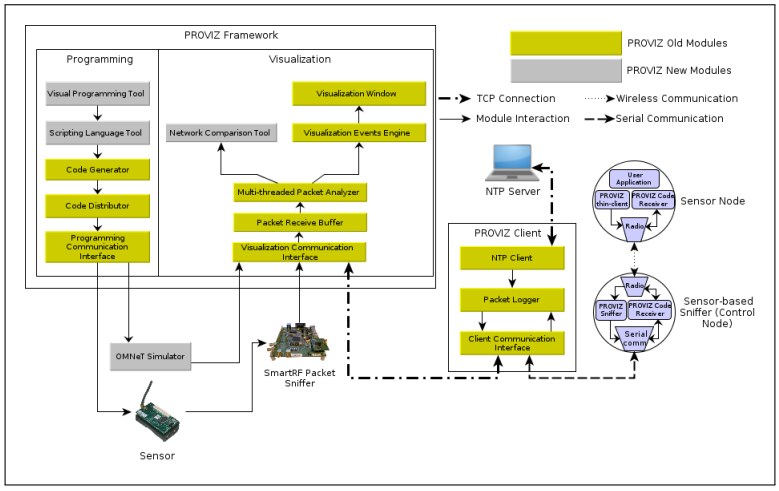


Figure 24: Overview of PROVIZ framework with new modules.

The tools can be used synchronously with each other to program, simulate or debug a WSN environment. Programs can be developed in the visualization tool or scripting language tool based on user's programming experience. The generated TinyOS code can be

burnt onto a real sensor or simulated in the OMNeT simulation environment. In this way, a heterogeneous WSN can be programmed or simulated. Sniffers can be deployed across the network and the visualization tool can be used to visualize packets in the WSN. Or, packet traces can be obtained from OMNeT or the SmartRF Packet Sniffer and passive visualization can be done with the visualization tool. If two networks are to be compared, the packet traces can be fed to the network comparison tool to generate a visual comparison of two WSNs. Figure 25 pictorially describes these tools working in sync.

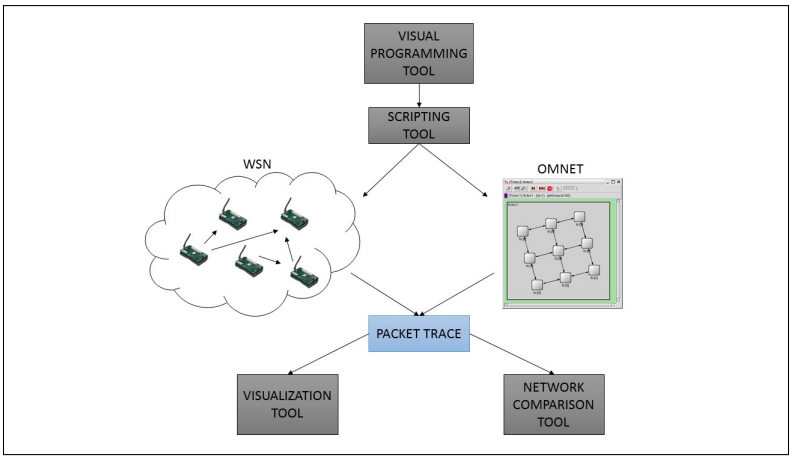


Figure 25: Tools in the PROVIZ framework working in sync.

CHAPTER VII

CONCLUSION AND FUTURE WORK

7.1 Research Contribution

In this thesis, three tools for a user-friendly programming framework for WSNs were developed. The network comparison tool enables researchers to compare two network deployments for differences in link characteristics such as average packet delay, number of bytes in payload, etc. This tool can help take programming concepts to K-12 students. It will aid teachers unfamiliar with programming to grade WSN programming assignments. The scripting language for TinyOS [1] is a C based language that offers application development for TinyOS in a single file, eliminating the wiring and interface concept of TinyOS. It has a scalable design where more add-on modules for projects like data collection, monitoring, routing, etc. can be programmed in TinyOS and abstracted to a C function. The tool also allows programmers to reuse WSN functions developed across files. The visual programming tool has simple drag-and-drop icons to create WSN programs and connect sensors via the radio interface. Again, this tool can help teach students basic programming concepts. These tools were developed on top of the PROVIZ framework [8], which is an open-source, platform-independent visualization tool for WSNs. All of these tools are remotely connected to the OMNeT simulator [11] that can simulate the programs created in PROVIZ and generate packet traces for visualization and comparison.

7.2 Future Research Direction

The tools will be further developed to encompass many WSN scenarios. The network comparison tool will be improved to draw differences between routes adopted by packets and filter routing messages in calculating link characteristics. The scripting language will be updated with more add-on modules for commonly used functions such as routing, data collection, etc. The visual programming tool will be extended to include more WSN operations and input and output operators. Users will be able to specify routing modules for sensor

communication. The PROVIZ visualization tool will also be extended from current drag-and-drop visualization to automatic detection of nodes in WSN visualization. An option to save sniffed packet data for later visualization or comparison will also be added.

APPENDIX A

PROVIZ HOW-TO

This section has detailed instructions on how to navigate through the various tools in PROVIZ. PROVIZ is developed using Qt Creator. Install Qt (4.7.4) and the graphviz libraries for smooth graphics generation. Install tinyos-2.x and tinyos-1.x libraries in the system (Ubuntu 12.10). Also, install OMNeT (4.7.4) for simulation of WSNs.

A.1 Starting PROVIZ

Open Qt Creator and the test1.pro PROVIZ file. Press the Run button to start PROVIZ.

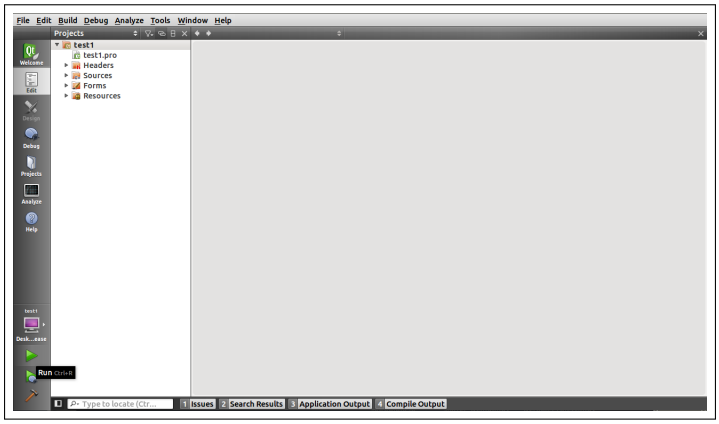


Figure 26: Start PROVIZ.

Press the clear button to reorganize the widgets in the window.

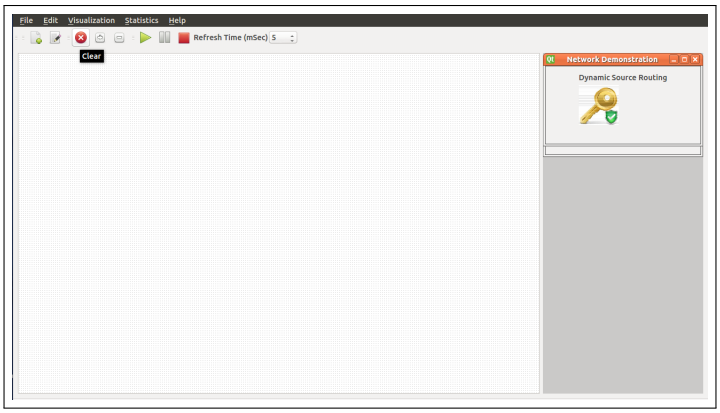


Figure 27: Reorganize widgets - BEFORE.

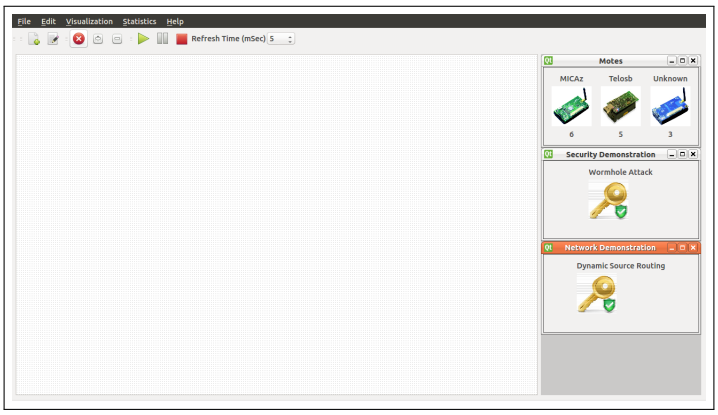


Figure 28: Reorganize widgets - AFTER.

A.2 Live Visualization

Program a sensor with the TinyOS BaseStation15.4 application and connect it to the system. Note the USB interface of the sensor. For example, if the application is installled using /dev/ttyUSB0, the serial input given to the sniffer program will be through /dev/ttyUSB1. Install the WSN application on the other sensors and start the live visualization tool in PROVIZ.

Drag and drop the sensors onto the canvas based on their type and give them the node IDs that they have been programmed with.

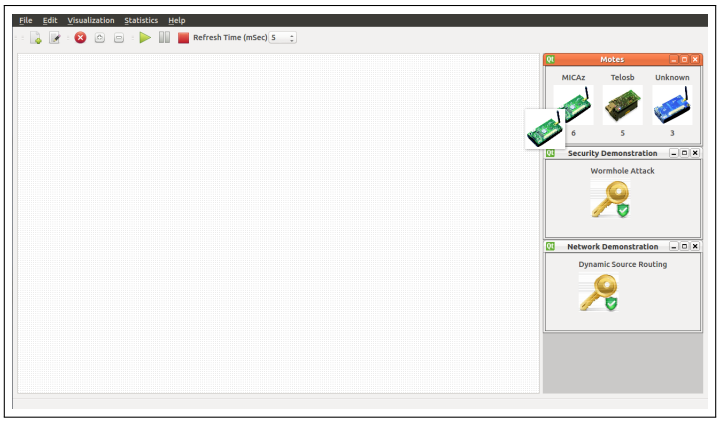


Figure 29: Drag and drop nodes.

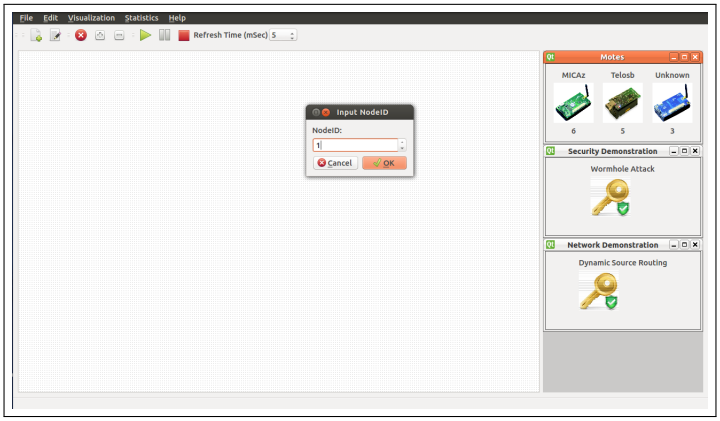


Figure 30: Set node IDs.

Once the nodes have been set up on the canvas, press the play button and choose live visualization.

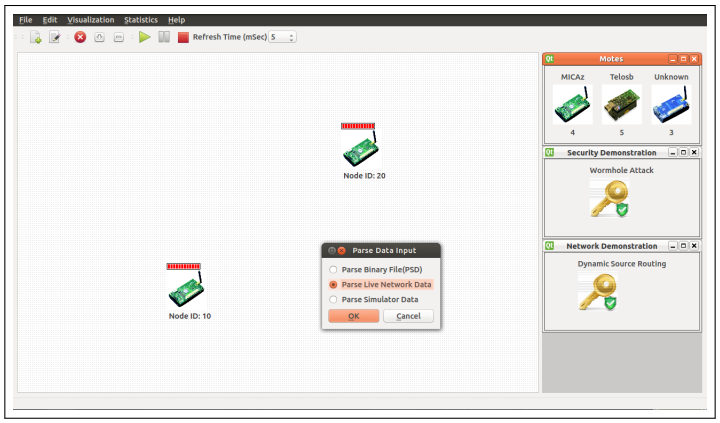


Figure 31: Choose live visualization option.

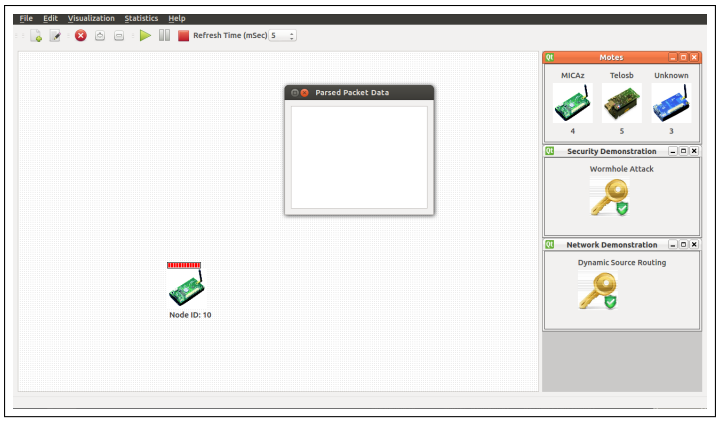


Figure 32: Program waits for input from sniffer.

Open the serial.pro project in Qt Creator and run the program. Give the following inputs, changing only the USB interface based on your sensor. This program feeds input from sniffer to the PROVIZ program.

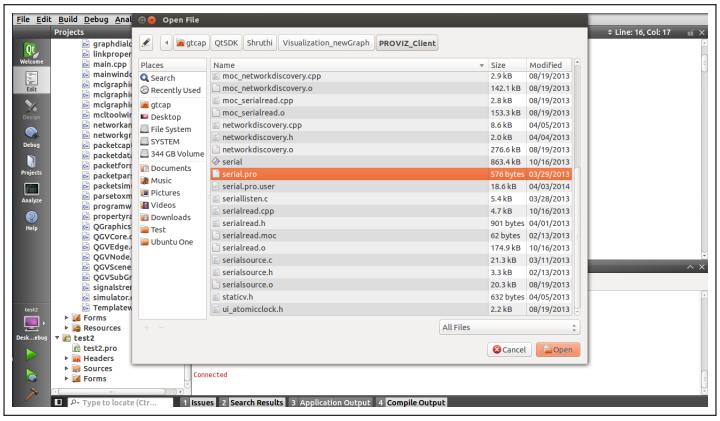


Figure 33: Open serial.pro project.

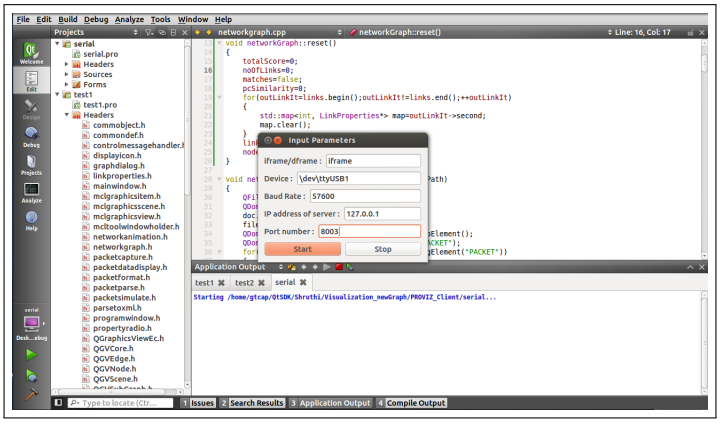


Figure 34: Set sniffer configuration and press Start.

Switch back to the live visualization window and watch the packet transmission in action.

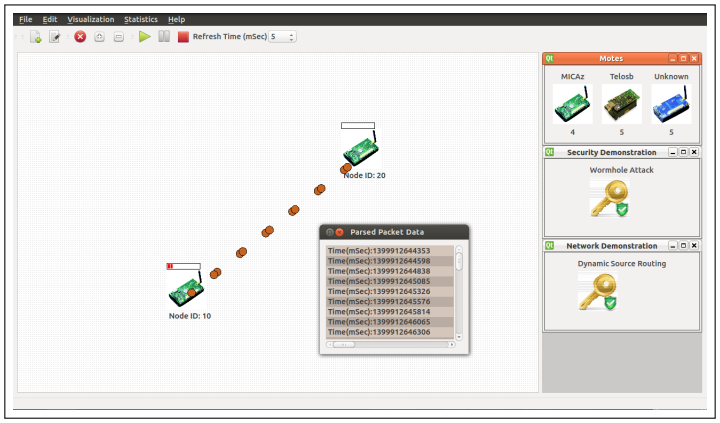


Figure 35: Live visualization of packet transmission.

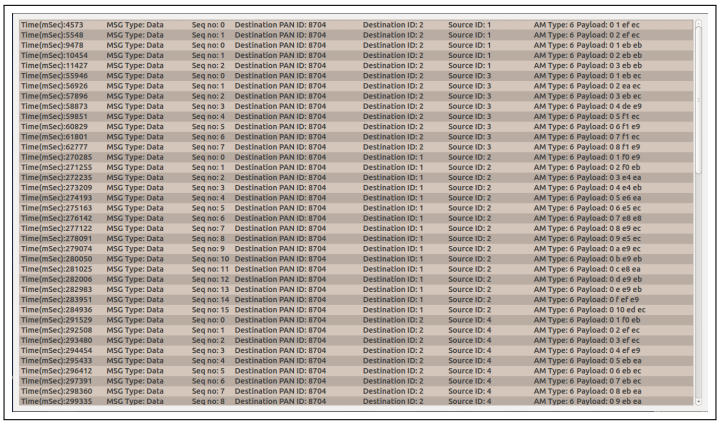


Figure 36: Expand the Parsed packet data window to see packet details.

A.3 Passive Visualization

Use the TI SmartRF Packet Sniffer and generate a .psd trace file of your WSN. Similar to live visualization, drag and drop sensors onto canvas with respective node IDs. Choose the Parse Binary File (PSD) option on pressing play and choose the trace file to be visualized.

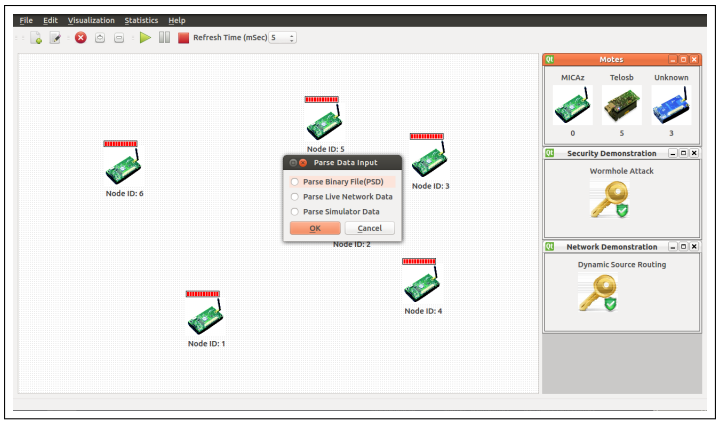


Figure 37: Choose Parse Binary File option.

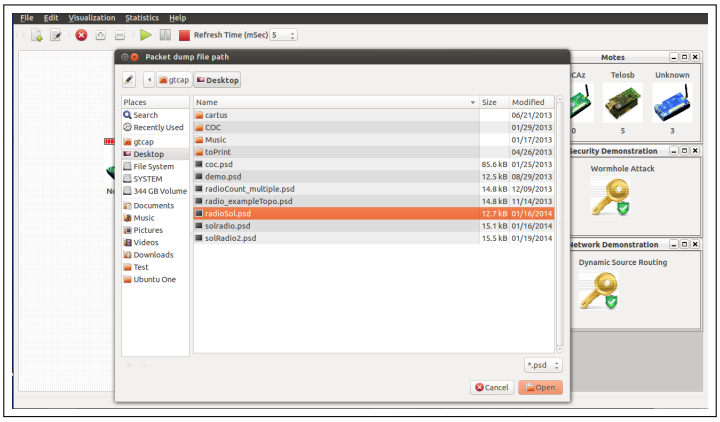


Figure 38: Choose the file to be visualized.

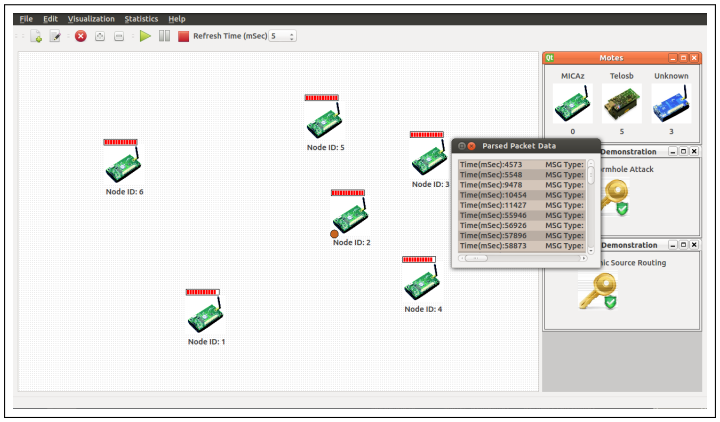


Figure 39: Packets visualized and output generated in parse data window.

A.4 Scripting tool

Open the script editor window from PROVIZ and write code using scripting language or open an existing code.

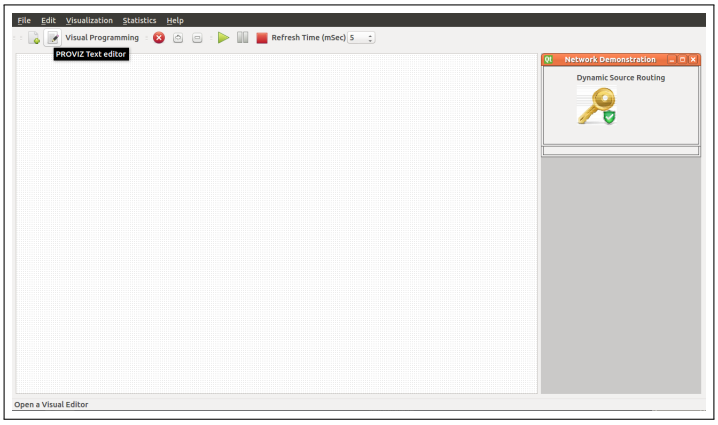


Figure 40: Open script editor window.

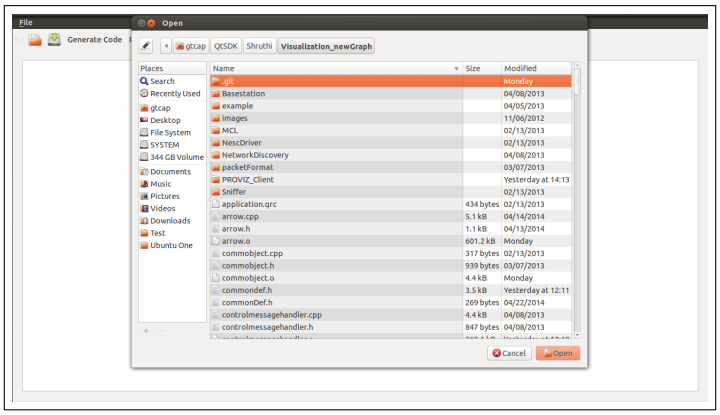


Figure 41: Open a pre-exisitng file.

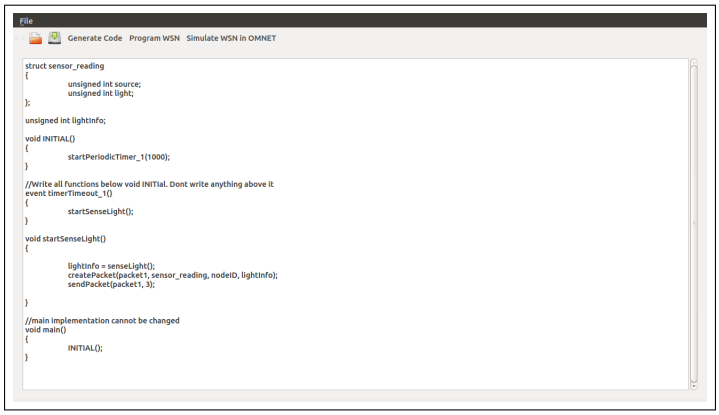


Figure 42: Code for sample application in scripting langauge.

Once the code is typed or opened from an external source, generate the TinyOS code for the same by pressing the generate code button. Then, press the program WSN button if the application is to be programmed on sensors or choose simulate in OMNeT option if resources are unavailable.

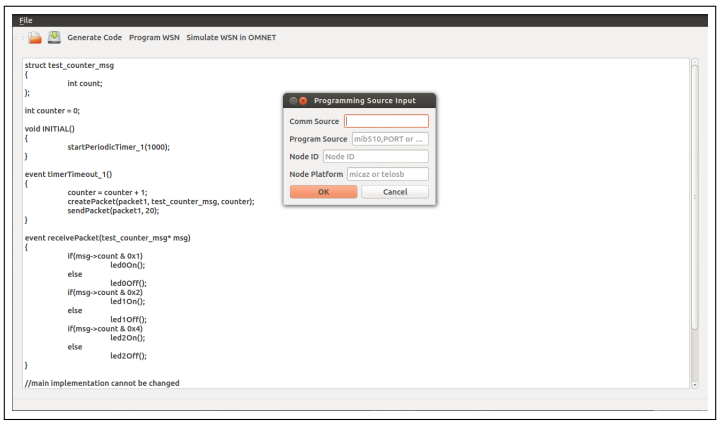


Figure 43: Set options for programming in WSN.

Give an application name for the program generation in OMNeT.

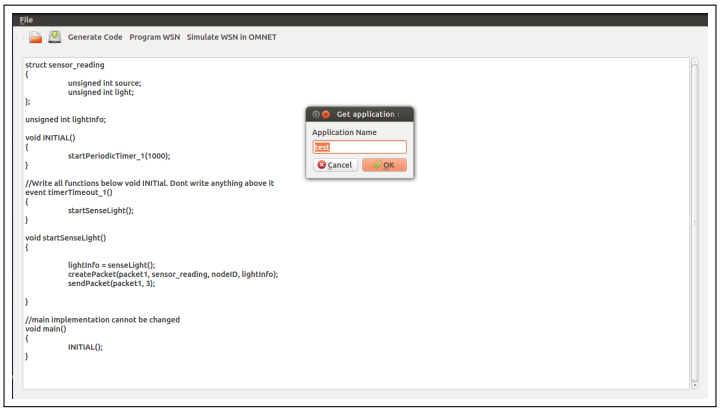


Figure 44: Application name for OMNeT.

If a heterogeneous WSN is to be programmed and simulated in OMNeT, give the number of nodes, node ID and application name for each node in the network and also, give the WSN name. This name will be used to visualize the WSN.

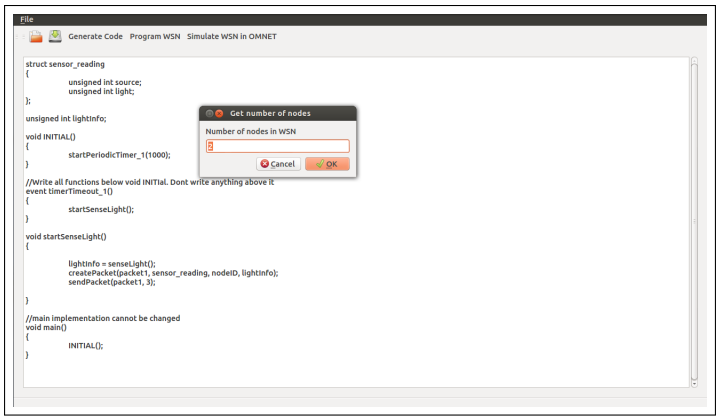


Figure 45: Set number of nodes in WSN.

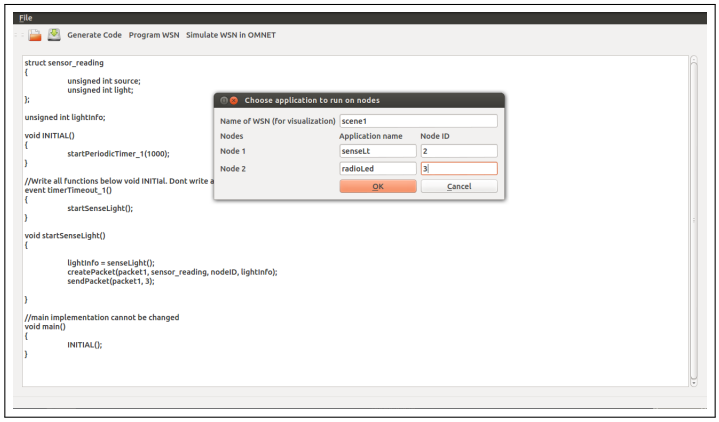


Figure 46: Set WSN name and node ID and application name for each node.

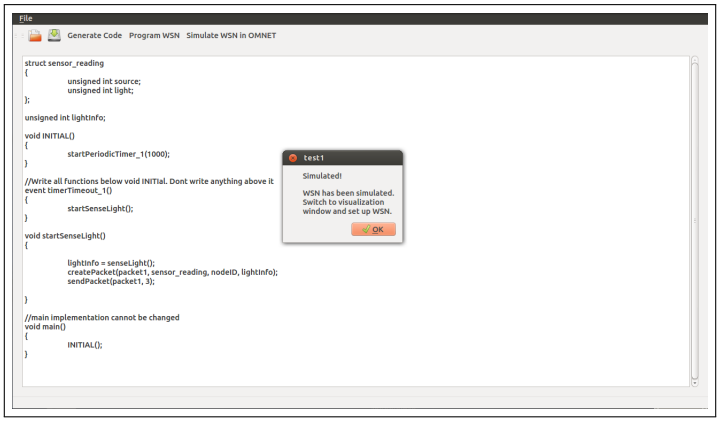


Figure 47: Message generated when simulation is over.

Switch back to visualization window to start visualization of OMNeT. Drag and drop nodes with node IDs and choose Parse Simulator Data option on pressing Play and give the WSN name to be simulated.

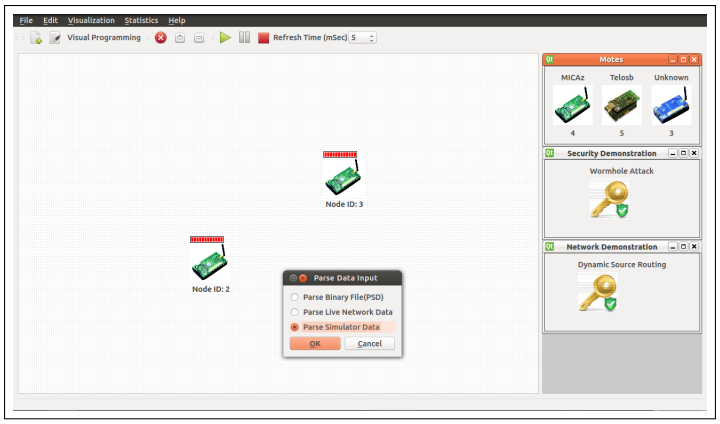


Figure 48: Choose Parse Simulator Data for OMNeT visualization.

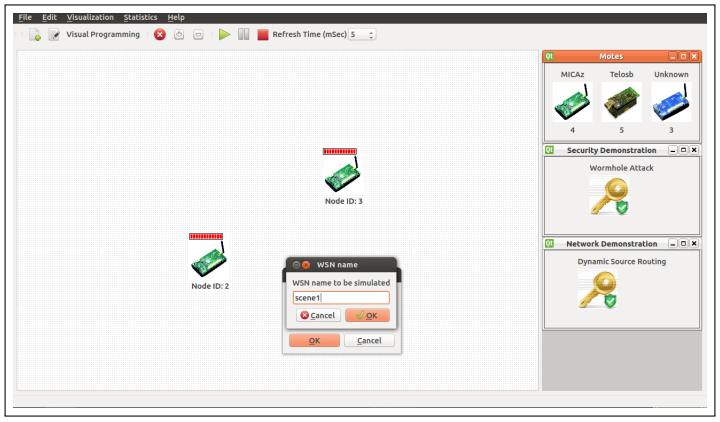


Figure 49: Provide WSN name to be simulated.

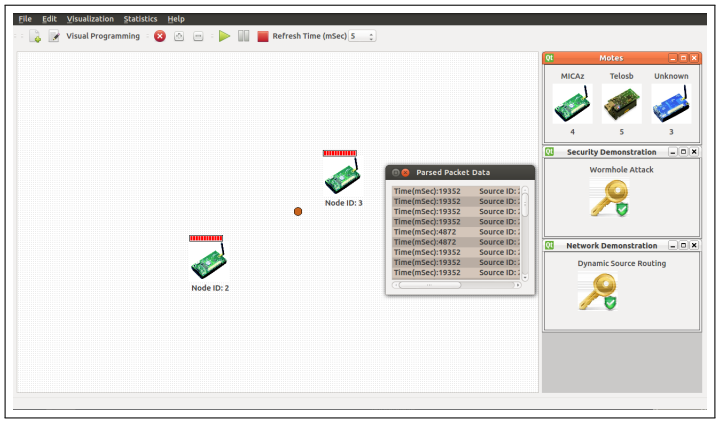


Figure 50: Visualization of OMNeT simulation packets.

A.5 Visual Programming

Switch to visual programming window in PROVIZ. The steps to navigate through visual programming are explained in chapter 5.

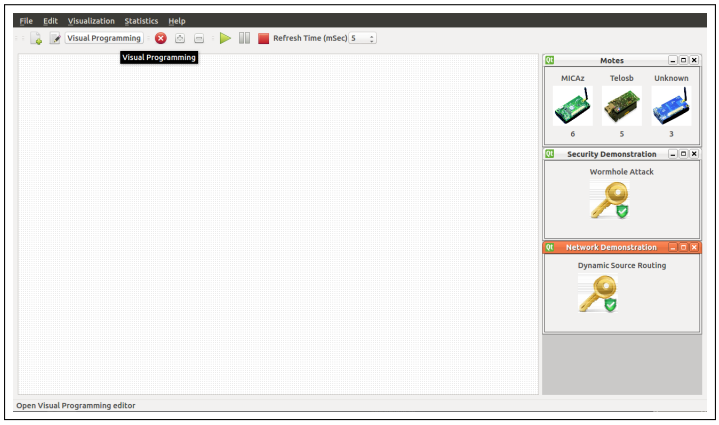


Figure 51: Switch to visual programming.

A.6 Demo Scene Visualization

Drag and drop the demo icons situated on the right, below the sensor list, to the canvas and hit play for visualization.

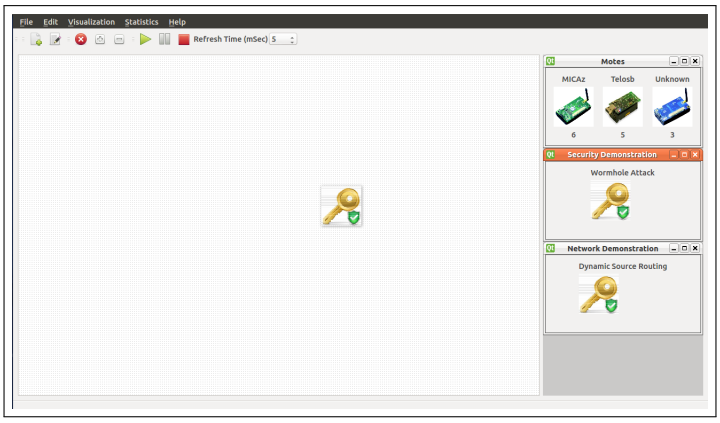


Figure 52: Drag and drop the demo icon.

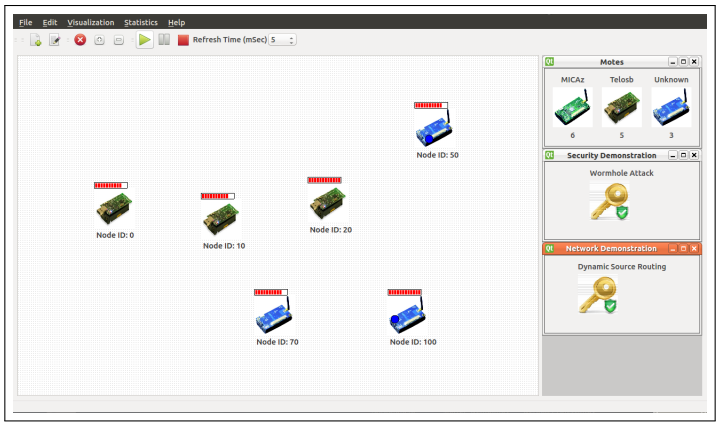


Figure 53: Press play and watch demo in action.

REFERENCES

[1] "TinyOS documentation." `http://docs.tinyos.net/`.

[2] A. Burns, B. R. Greene, M. J. McGrath, T. J. OShea, B. Kuris, S. M. Ayer, F. Stroiescu, and V. Cionca, "SHIMMER – a wireless sensor platform for noninvasive biomedical research," *IEEE Sensor Journal*, vol. 10, no. 9, pp. 1527–1534, 2010.

[3] N. Kurata, B. Spencer, and M. Ruiz-Sandoval, "Risk monitoring of buildings with wireless sensor networks," *Structural Control and Health Monitoring*, vol. 12, no. 3-4, pp. 315–327, 2005.

[4] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesc language: A holistic approach to networked embedded systems," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 1–11, 2003.

[5] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica, "The design and implementation of a declarative sensor network system," in *Proceedings of the 5th ACM International Conference on Embedded Networked Sensor Systems*, pp. 175–188, 2007.

[6] B. Greenstein, E. Kohler, and D. Estrin, "A sensor network application construction kit (SNACK)," in *Proceedings of the 2nd ACM International Conference on Embedded Networked Sensor Systems*, pp. 69–80, 2004.

[7] L. Ma, L. Wang, L. Shu, J. Zhao, S. Li, Z. Yuan, and N. Ding, "Netviewer: a universal visualization tool for wireless sensor networks," in *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM)*, pp. 1–5, 2010.

[8] R. Chandrasekar, S. Uluagac, and R. Beyah, "PROVIZ: An integrated visualization and programming framework for WSNs," in *Proceedings of the IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp)*, pp. 146–149, 2013.

[9] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[10] "Alice." `www.alice.org/`.

[11] "OMNET Simulator." `http://www.omnetpp.org/`.

[12] M. Turon and J. Suh, "MOTE–VIEW: a sensor network monitoring and management tool," in *Proceedings of the 2nd IEEE workshop on Embedded Networked Sensors*, pp. 11–17, 2005.

[13] "MEMSIC." `www.memsic.com/`.

[14] B.-R. Chen, G. Peterson, G. Mainland, and M. Welsh, "Livenet: Using passive monitoring to reconstruct sensor network dynamics," in *Proceedings of Springer Distributed Computing in Sensor Systems*, pp. 79–98, 2008.

[15] M. Ringwald, K. Rmer, and A. Vitaletti, "SNIF: Sensor network inspection framework," Tech. Rep. 535, ETH Zurich, Institute for Pervasive Computing, 2006.

[16] N. Ramanathan, E. Kohler, L. Girod, and D. Estrin, "Sympathy: a debugging system for sensor networks [wireless networks]," in *Proceedings of the 9th Annual IEEE International Conference on Local Computer Networks*, pp. 554–555, 2004.

[17] M. Dyer, J. Beutel, T. Kalt, P. Oehen, L. Thiele, K. Martin, and P. Blum, "Deployment support network," in *Springer Wireless Sensor Networks*, pp. 195–211, 2007.

[18] R. Jurdak, A. G. Ruzzelli, A. Barbirato, and S. Boivineau, "Octopus: monitoring, visualization, and control of sensor networks," *Wiley Wireless communications and Mobile computing*, vol. 11, no. 8, pp. 1073–1091, 2011.

[19] F. SantAnna, N. d. L. R. Rodriguez, and R. Ierusalimschy, "Ceu: Embedded, safe, and reactive programming," *PUC-Rio, Tech. Rep*, vol. 12, p. 12, 2012.

[20] "Datalog: Deductive database programming." `docs.racket-lang.org/datalog/`.

[21] E. Cheong, E. A. Lee, and Y. Zhao, "Viptos: a graphical development and simulation environment for tinyOS-based wireless sensor networks," in *Proceedings of the 3rd ACM International Conference on Embedded Networked Sensor Systems*, pp. 302–302, 2005.

[22] W. P. McCartney and N. Sridhar, "Tosdev: a rapid development environment for tinyOS," in *Proceedings of the 4th ACM International Conference on Embedded Networked Sensor Systems*, pp. 387–388, 2006.

[23] "Eclipse." `https://www.eclipse.org/`.

[24] J. B. Lim, B. Jang, S. Yoon, M. L. Sichitiu, and A. G. Dean, "RaPTEX: Rapid prototyping tool for embedded communication systems," *ACM Transactions on Sensor Networks*, vol. 7, no. 1, 2010.

[25] B. L. Titzer, D. K. Lee, and J. Palsberg, "Avrora: Scalable sensor network simulation with precise timing," in *Proceedings of the 4th IEEE International Symposium on Information Processing in Sensor Networks*, p. 67, 2005.

[26] F. Vieira, B. A. Vitorino, M. A. Vieira, D. Silva, A. Fernandes, and A. Loureiro, "WISDOM: a visual development framework for multi-platform wireless sensor networks," in *Proceedings of the 10th IEEE Conference on Emerging Technologies and Factory Automation*, vol. 2, 2005.

[27] A. Elsts, J. Judvaitis, and L. Selavo, "SEAL: A domain-specific language for novice wireless sensor network programmers," in *Proceedings of the 39th IEEE EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 220–227, 2013.

[28] "Blocky: A visual porgramming editor." `https://code.google.com/p/blockly/`.

[29] M. T. Hansen and B. Kusy, "Tinyinventor: A holistic approach to sensor network application development," *Extending the Internet to Low power and Lossy Networks*, 2011.

[30] R. V. Roque, *OpenBlocks: An extendable framework for graphical block programming systems*. PhD thesis, Massachusetts Institute of Technology, 2007.

[31] "Approved draft revision for ieee standard for information technology-telecommunications and information exchange between systems-local and metropolitan area networks-specic requirements-part 15.4b: Wire-less medium access control (mac) and physical layer (phy) specications for low rate wireless personal area networks (wpans) (amendment of ieee std 802.15.4-2003)," IEEE Std P802.15.4/D6, 2006.

[32] "TI packet sniffer." `http://www.ti.com/tool/packet-sniffer`.

[33] "Graphviz – graph visualization software." `http://www.graphviz.org/`.

[34] "Interactive qt graphviz display." `http://code.google.com/p/qgv/`.

[35] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: Accurate and scalable simulation of entire tinyOS applications," in *Proceedings of the 1st ACM international conference on Embedded networked sensor systems*, pp. 126–137, 2003.

[36] "Nesct: A language translator." `http://nesct.sourceforge.net/`.

[37] "Qt: A GUI development framework." `http://qt-project.org/`.

[38] M. Valero, S. Uluagac, S. Venkatachalam, K. Ramalingam, and R. Beyah, "The monitoring core: A framework for sensor security application development," in *Proceedings of the IEEE 9th International Conference on Mobile Adhoc and Sensor Systems (MASS)*, pp. 263–271, 2012.